Simplifying let, variables, operators, and if expressions

Simplification

Workspace

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

Stack

Heap

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Simplification

Workspace

```
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

Stack

Heap

```
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Simplification

Workspace

```
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

x is not a value: so look it up in the stack

Simplification

Workspace

```
let y = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

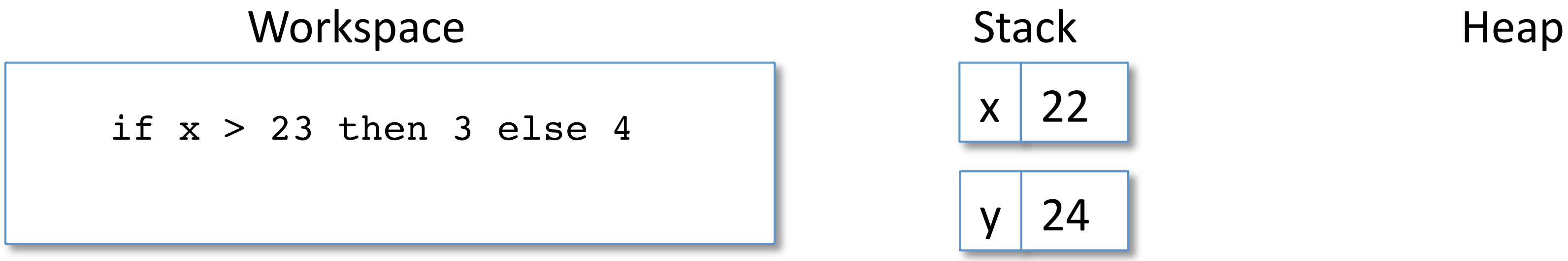
```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification



Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification

Workspace

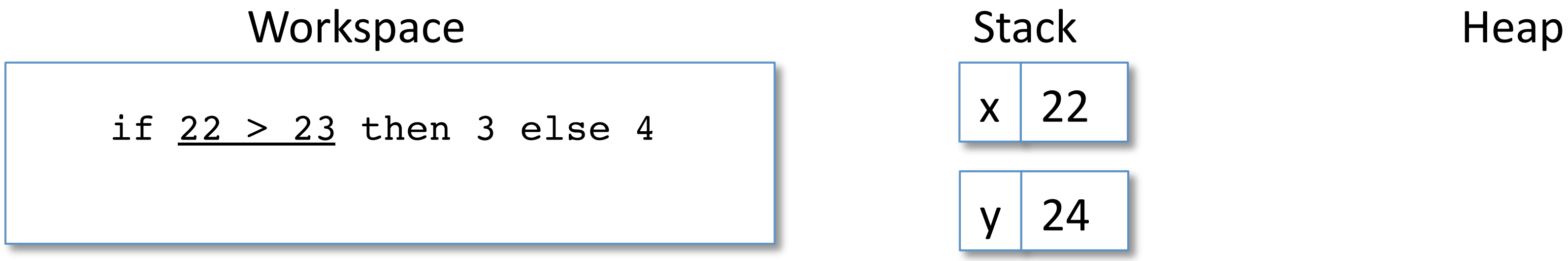
```
if 22 > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification

Workspace

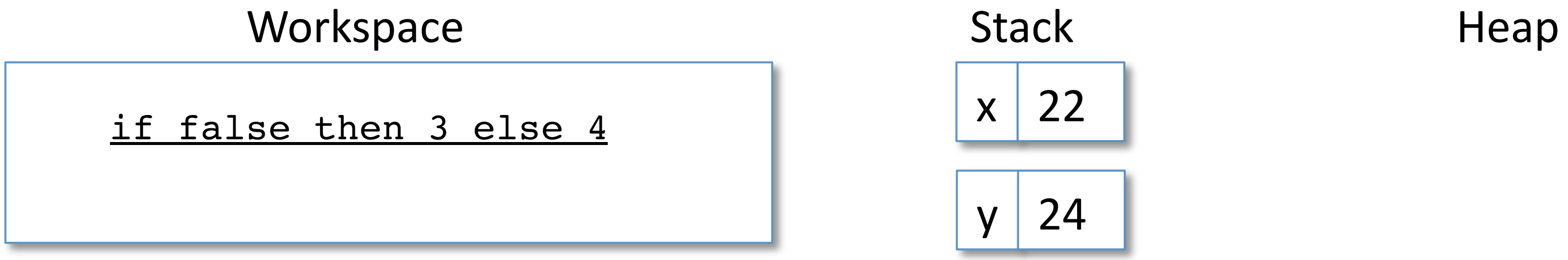
```
if false then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification



Workspace

```
if false then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification

Workspace

```
4
```

Stack

x	22
y	24

Heap

DONE!

Simplification Rules

- A let-expression "`let x = e in body`" is ready if the expression `e` is a *value*
  - it is simplified by adding a binding of `x` to `e` at the end of the stack and leaving body in the workspace
- A variable is always ready
  - it is simplified by replacing it with its value from the stack, where binding lookup goes in order from most recent to least recent
- A primitive operator (like +) is ready if both of its arguments are values
  - it is simplified by replacing it with the result of the operation
- An "if" expression is ready if the test is true or false
  - if it is true, it is simplified by replacing it with the then branch
  - if it is false, it is simplified by replacing it with the else branch

Simplifying lists and datatypes using the heap

Simplification

Workspace

```
1::2::3::[]
```

Stack

Heap

For uniformity, we’ll pretend lists are declared like this:

```
type ‘a list =
  | Nil
  | Cons of ‘a * ‘a list
```

Simplification

Workspace

Stack

Heap

```
Cons (1,Cons (2,Cons (3,Nil)))
```

For uniformity, we’ll pretend lists are declared like this:

```
type ‘a list =
 | Nil
 | Cons of ‘a * ‘a list
```

Simplification

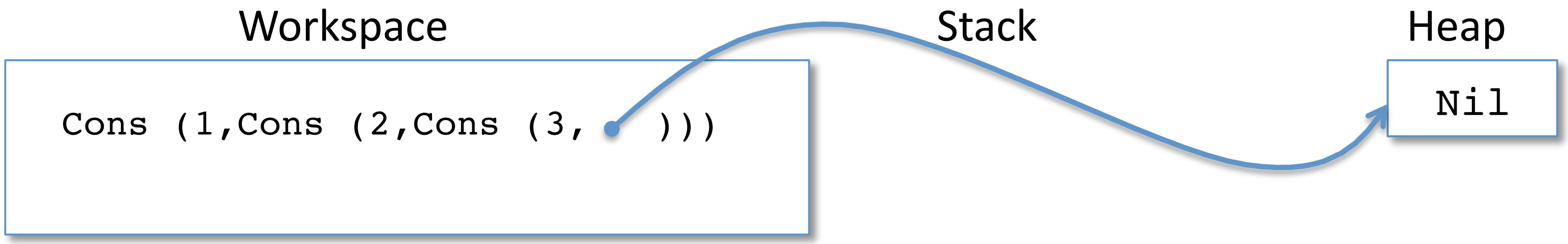
Workspace

Stack

Heap

```
Cons (1,Cons (2,Cons (3,Nil)))
```

Simplification



Workspace

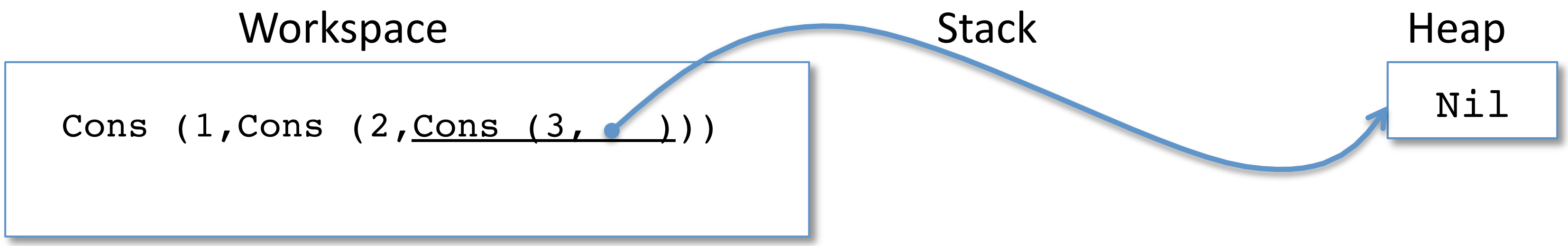
Stack

Heap

```
Cons (1,Cons (2,Cons (3,    )))
```

```
Nil
```

Simplification



Workspace

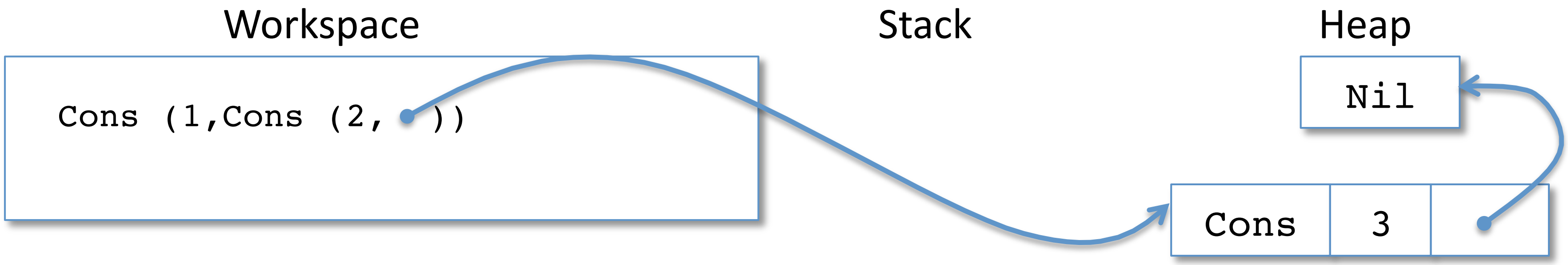
Stack

Heap

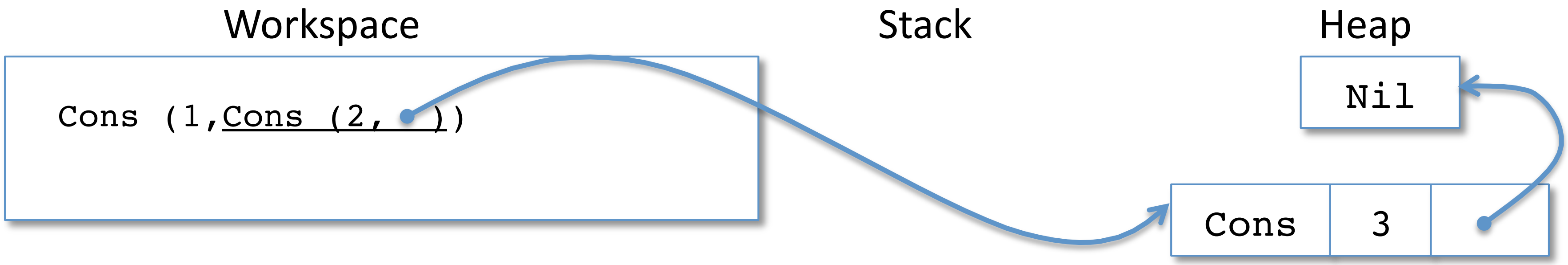
```
Cons (1,Cons (2,Cons (3,   )))
```

```
Nil
```

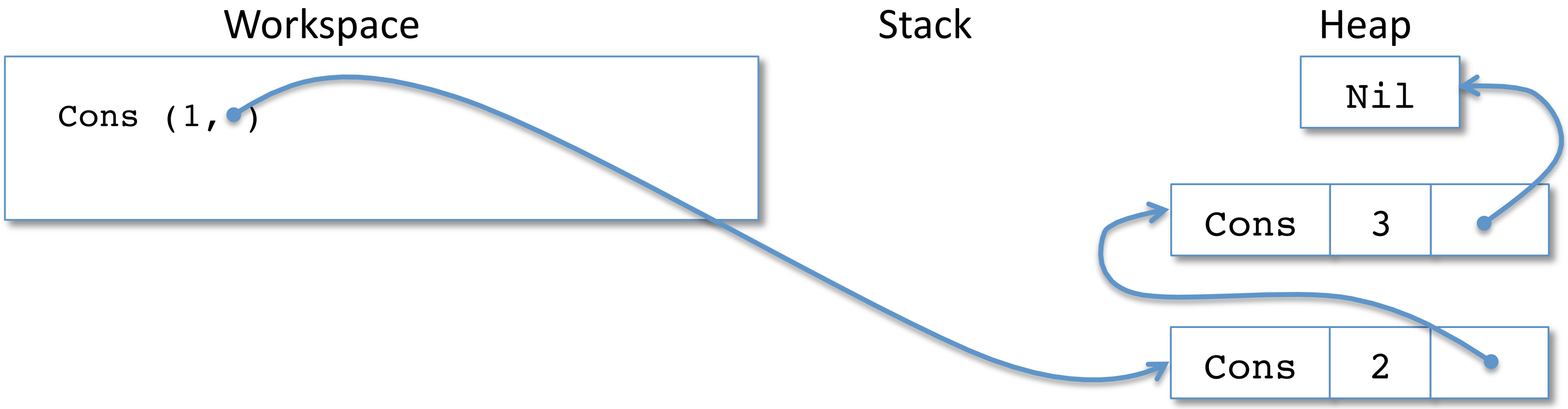
Simplification



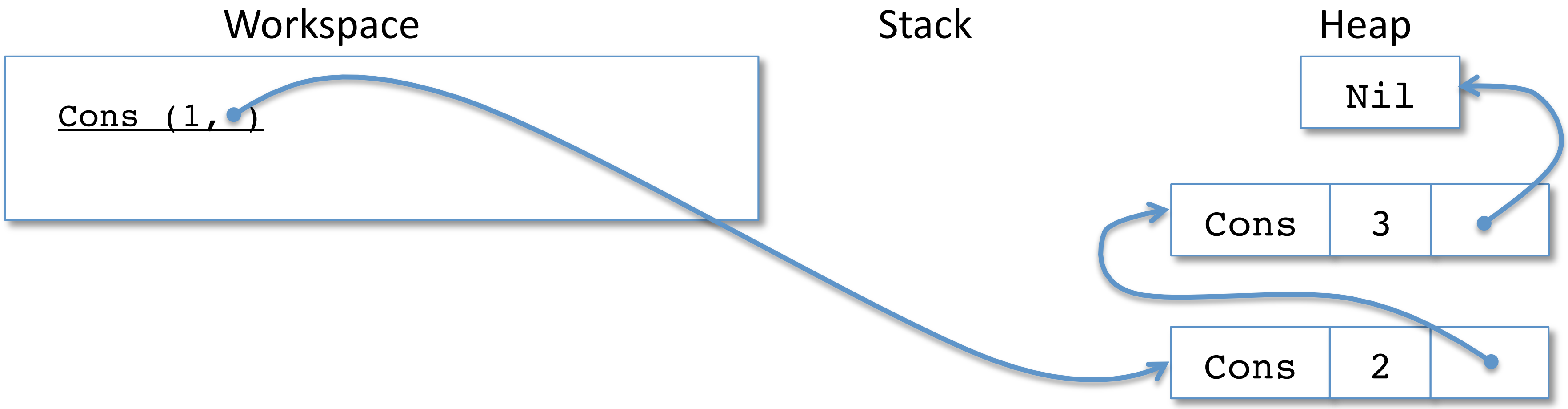
Simplification



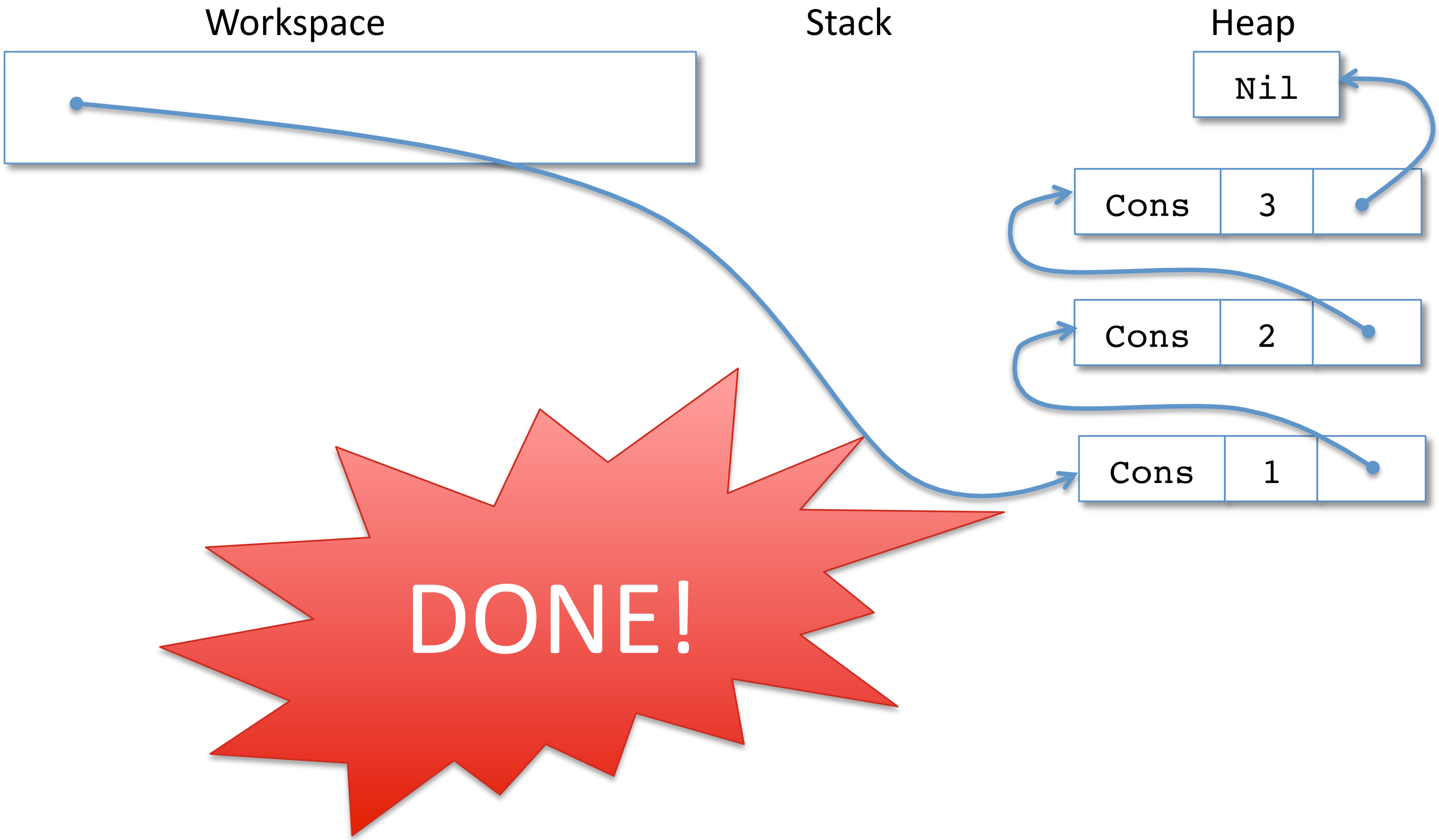
Simplification



Simplification



Simplification



Simplifying Datatypes

- A datatype constructor (like `Nil` or `Cons`) is ready if all its arguments are values
  - It is simplified by:
    - creating a new heap cell labeled with the constructor and containing the argument values*
    - replacing the constructor expression in the workspace by a reference to this heap cell

*Note: in OCaml, using a datatype constructor causes some space to be automatically allocated on the heap. Other languages have different mechanisms for accomplishing this: for example, the keyword 'new' in Java works similarly (as we'll see in a few weeks).

Simplifying functions

Function Simplification

Workspace

Stack

Heap

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Function Simplification

Workspace

Stack

Heap

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Function Simplification

Workspace

```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

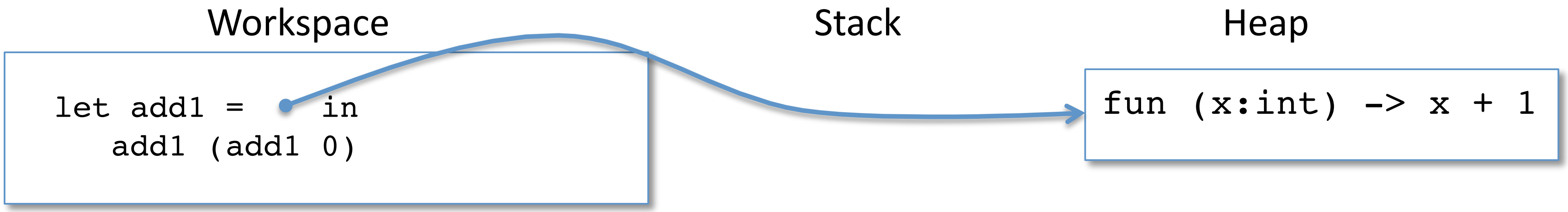
Workspace

Stack

Heap

```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

Function Simplification



Workspace

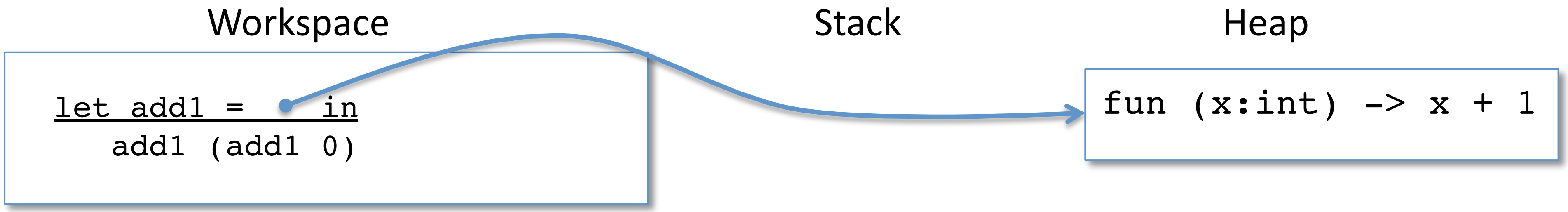
```
let add1 =    in
   add1 (add1 0)
```

Stack

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

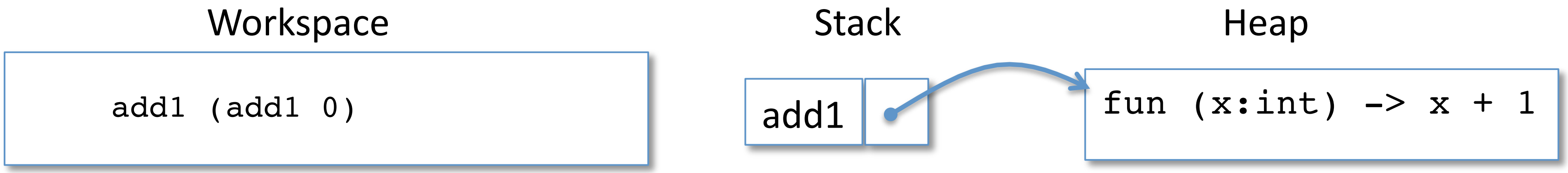
```
let add1 =     in
   add1 (add1 0)
```

Stack

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

```
add1 (add1 0)
```

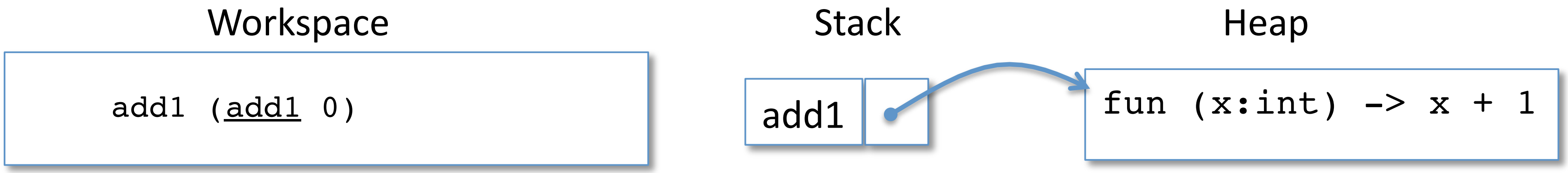
Stack

add1

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

```
add1 (add1 0)
```

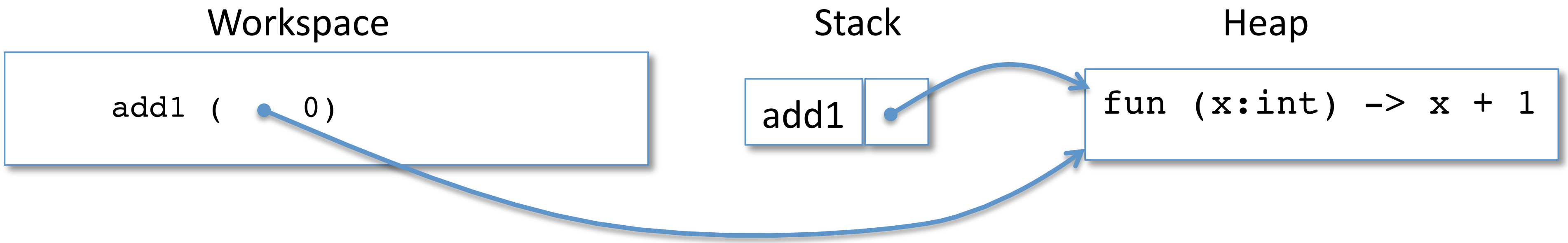
Stack

add1

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

`add1 (   0)`

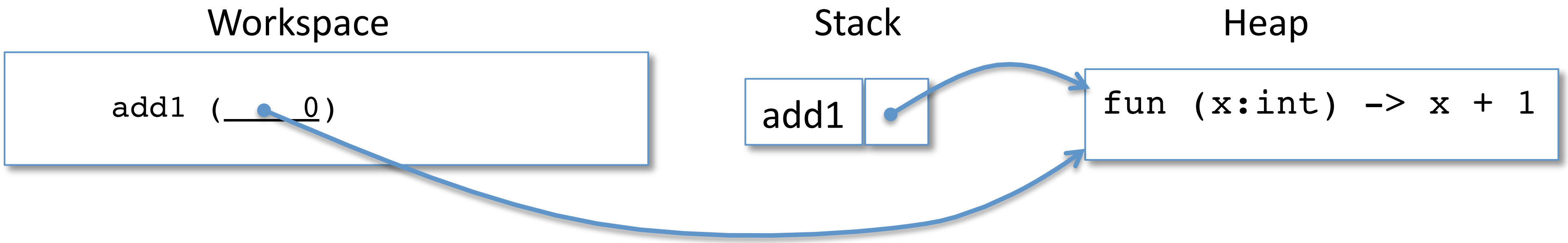
Stack

add1

Heap

`fun (x:int) -> x + 1`

Function Simplification



Workspace

```
add1 (   0)
```

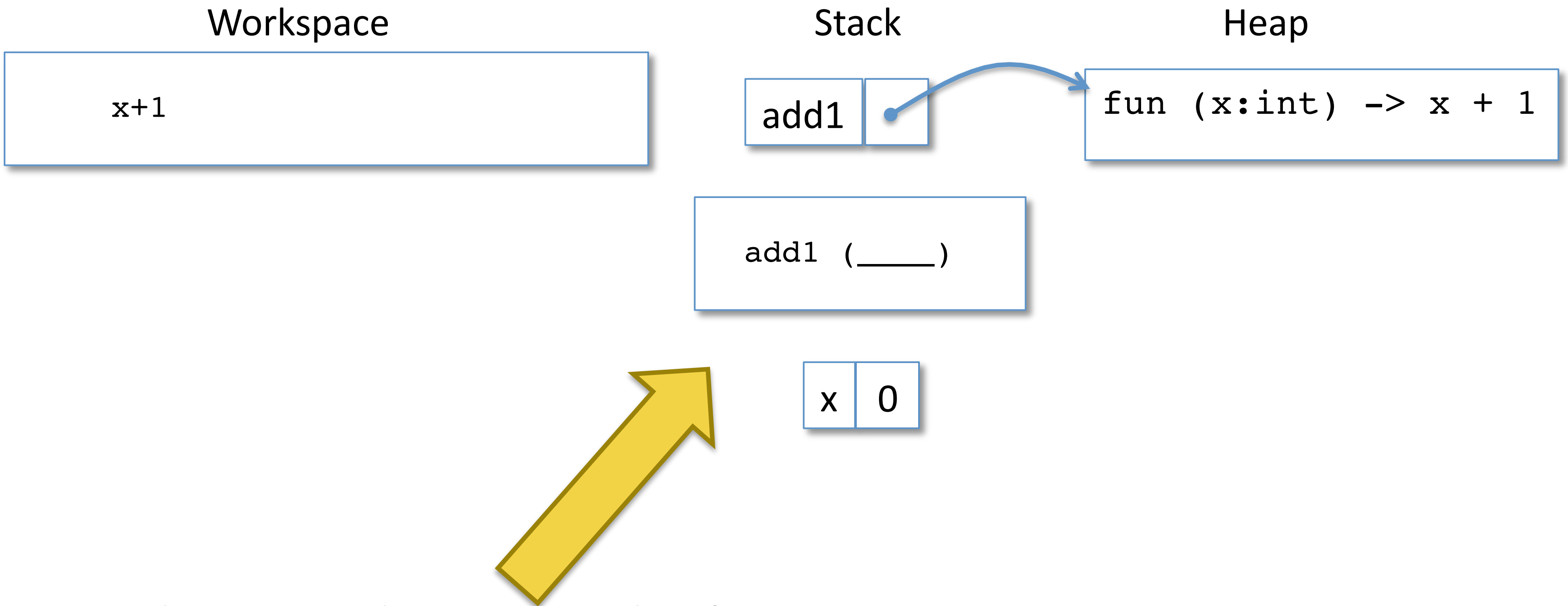
Stack

add1

Heap

```
fun (x:int) -> x + 1
```

Do the Call, Saving the Workspace


Workspace

Stack

Heap

```
x+1
```

add1

```
fun (x:int) -> x + 1
```

```
add1 (____)
```

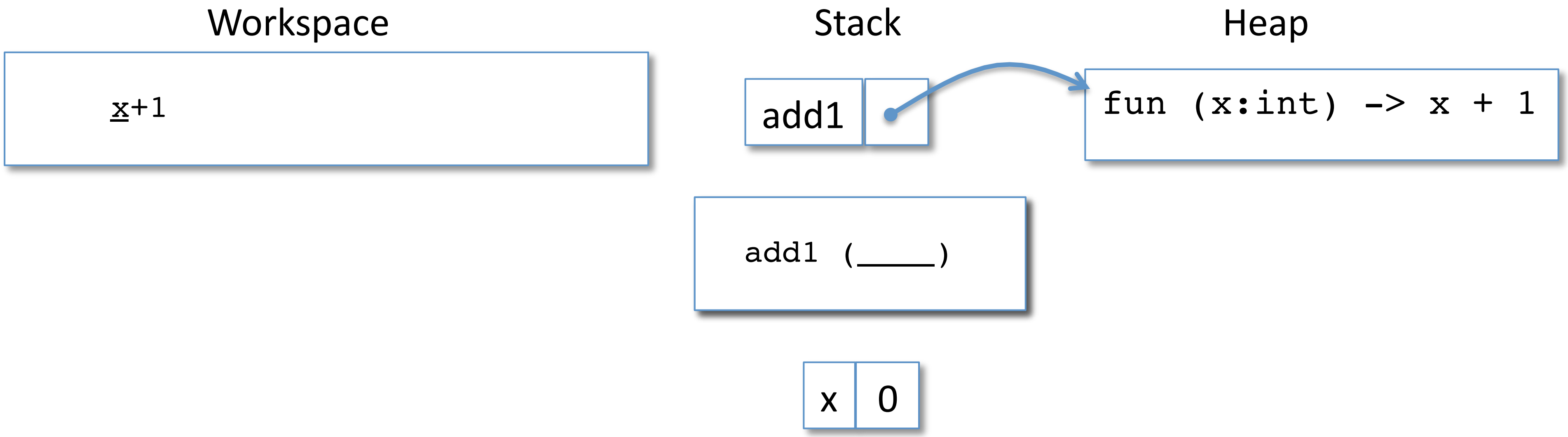
x 0

Note the saved workspace and pushed function argument.

- compare with the workspace on the previous slide.
- the name ‘x’ comes from the name in the heap

The new workspace is the *body* of the function

Function Simplification



Workspace

x+1

Stack

add1

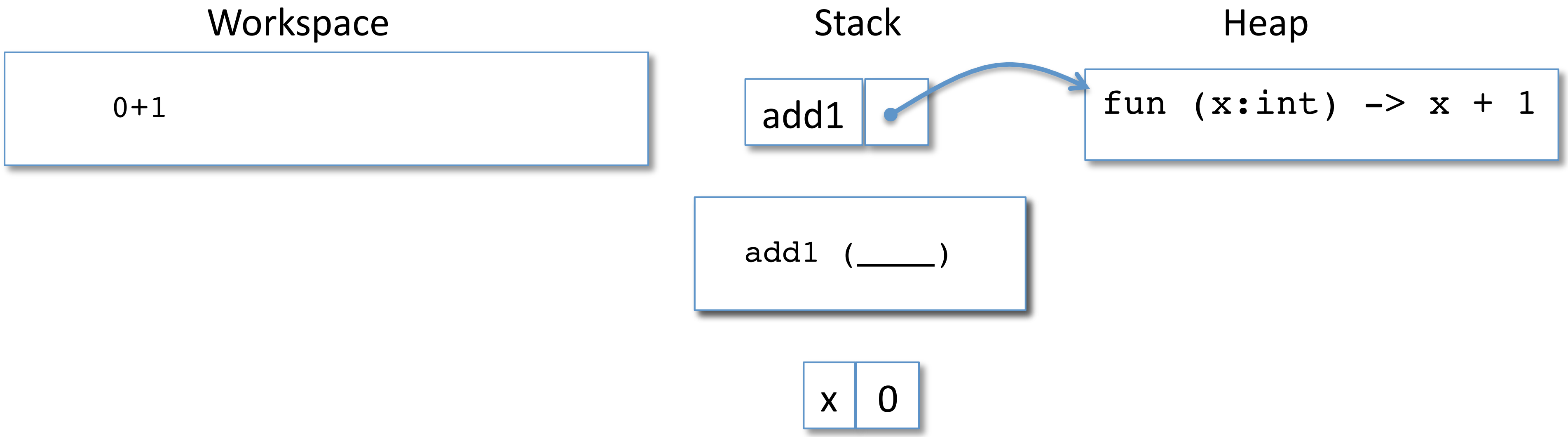
add1 (____)

x 0

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

```
0+1
```

Stack

add1

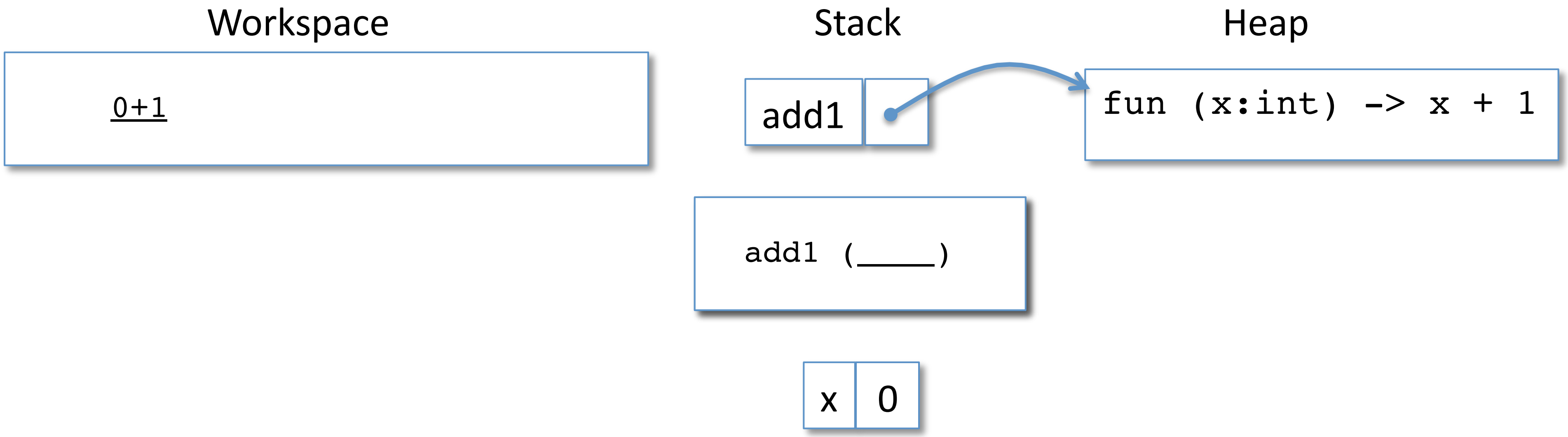
```
add1 (_____)
```

x 0

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

`0+1`

Stack

add1

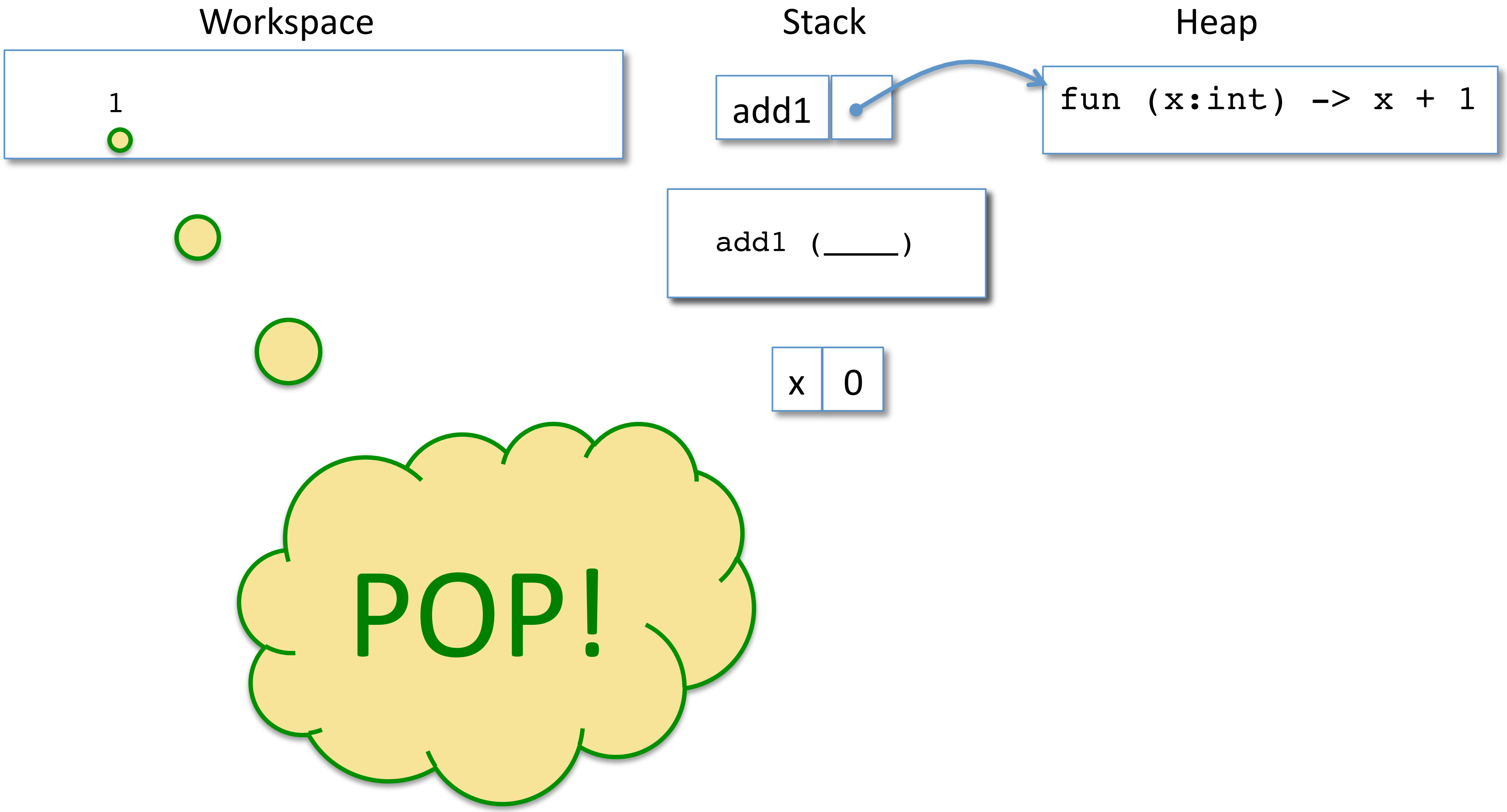
`add1 (____)`

x 0

Heap

`fun (x:int) -> x + 1`

Function Simplification



Workspace

1

Stack

add1

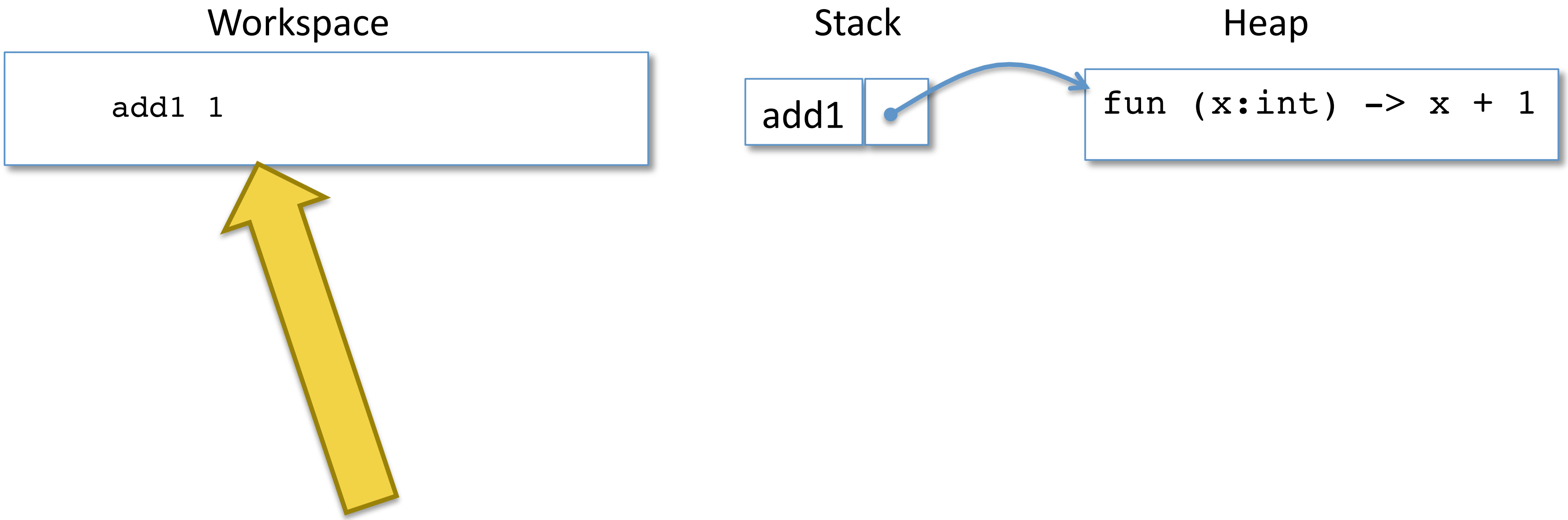
add1 (____)

x 0

Heap

`fun (x:int) -> x + 1`

Function Simplification



Workspace

```
add1 1
```

Stack

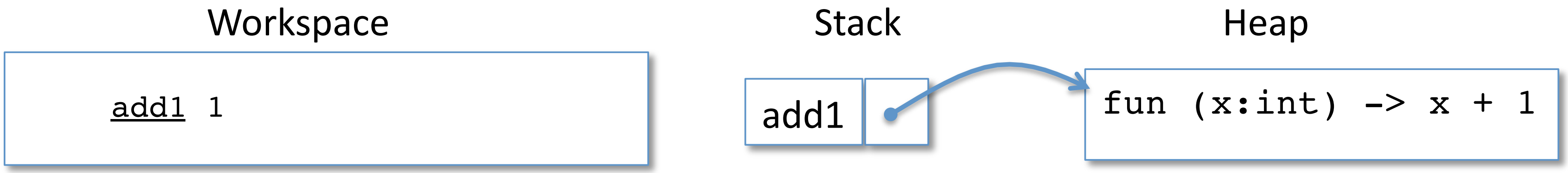
add1

Heap

```
fun (x:int) -> x + 1
```

See how the ASM *restored* the saved workspace, replacing its `hole’ with the value computed into the old workspace. (Compare with previous slide.)

Function Simplification



Workspace

add1 1

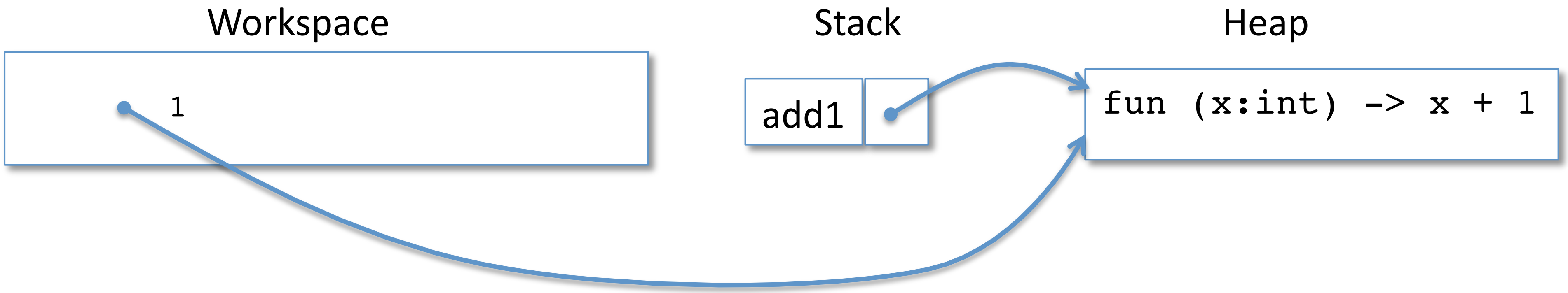
Stack

add1

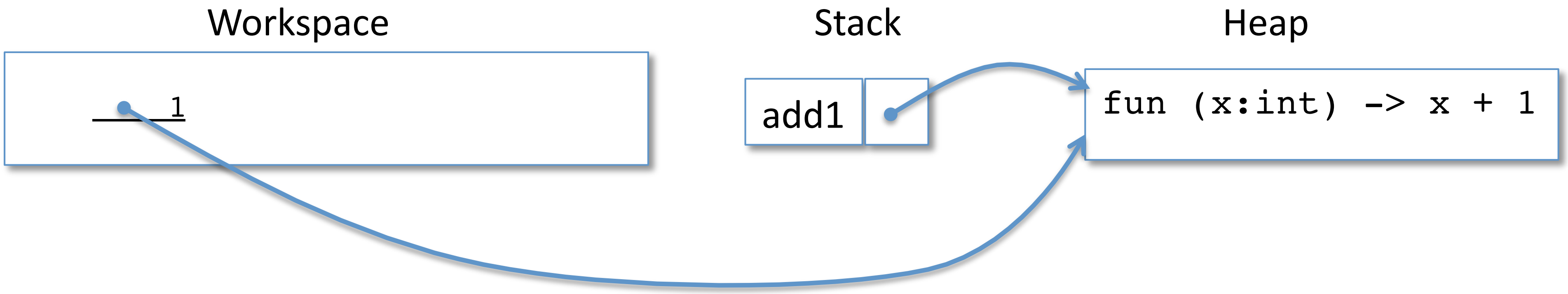
Heap

fun (x:int) -> x + 1

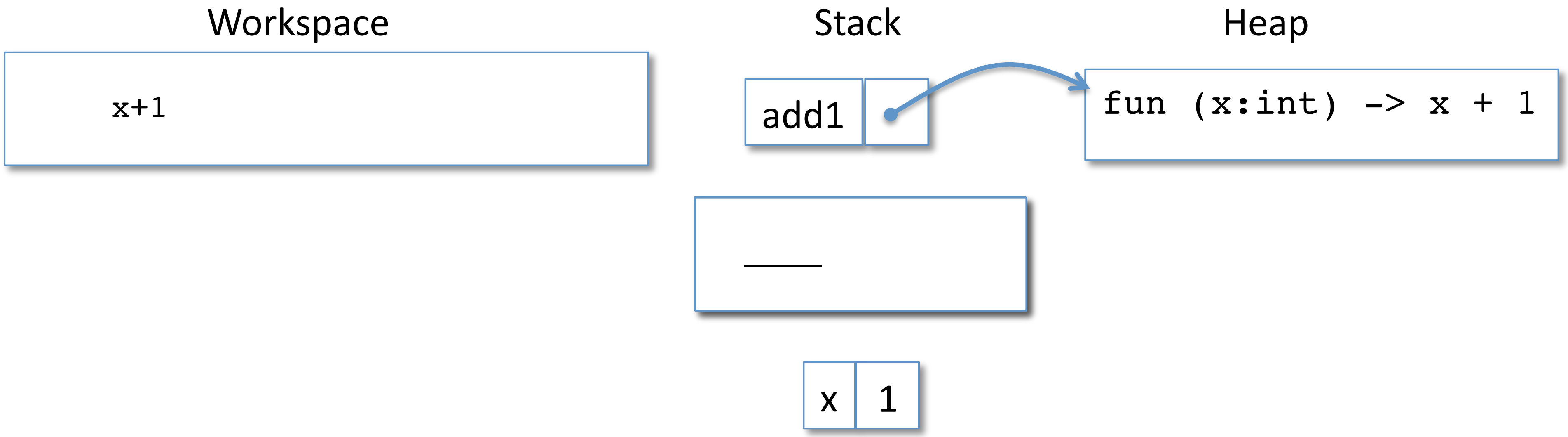
Function Simplification



Function Simplification



Function Simplification



Workspace

```
x+1
```

Stack

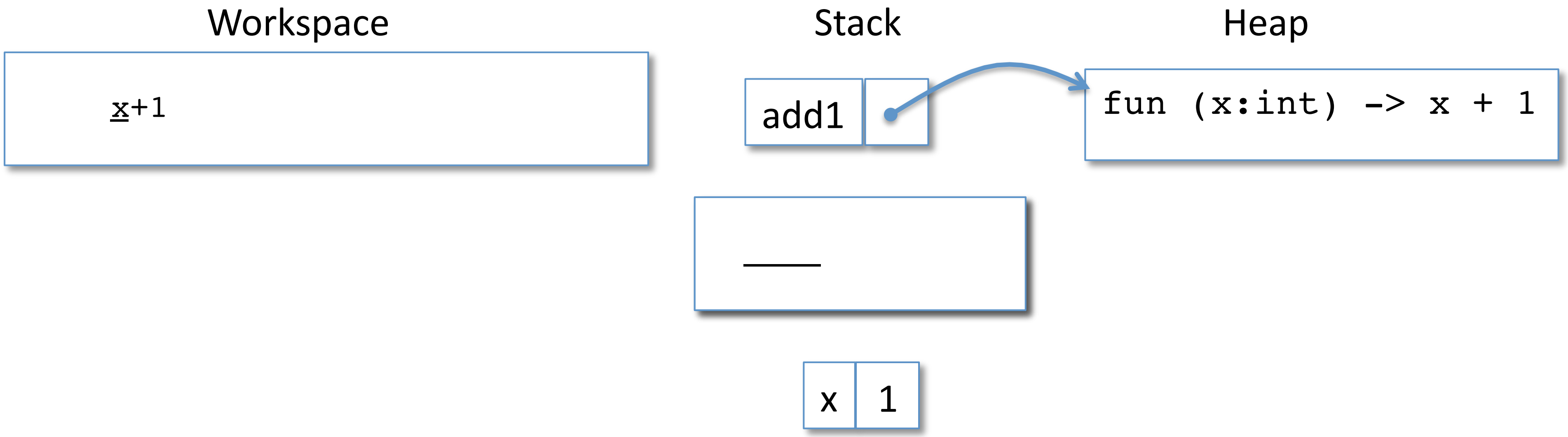
add1

x 1

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

```
x+1
```

Stack

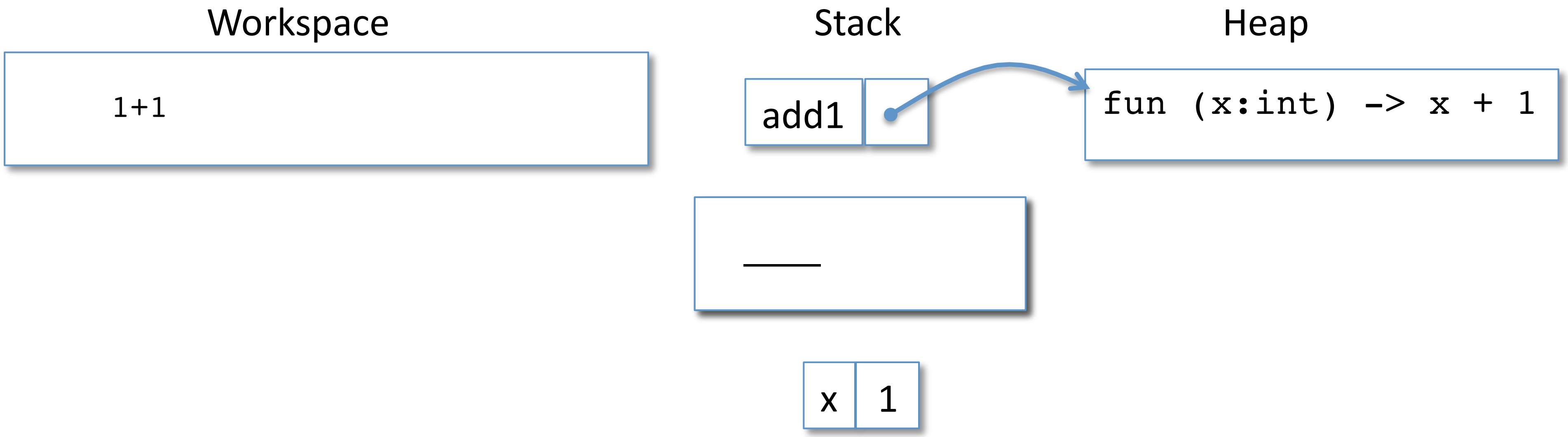
add1

x 1

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

```
1+1
```

Stack

add1

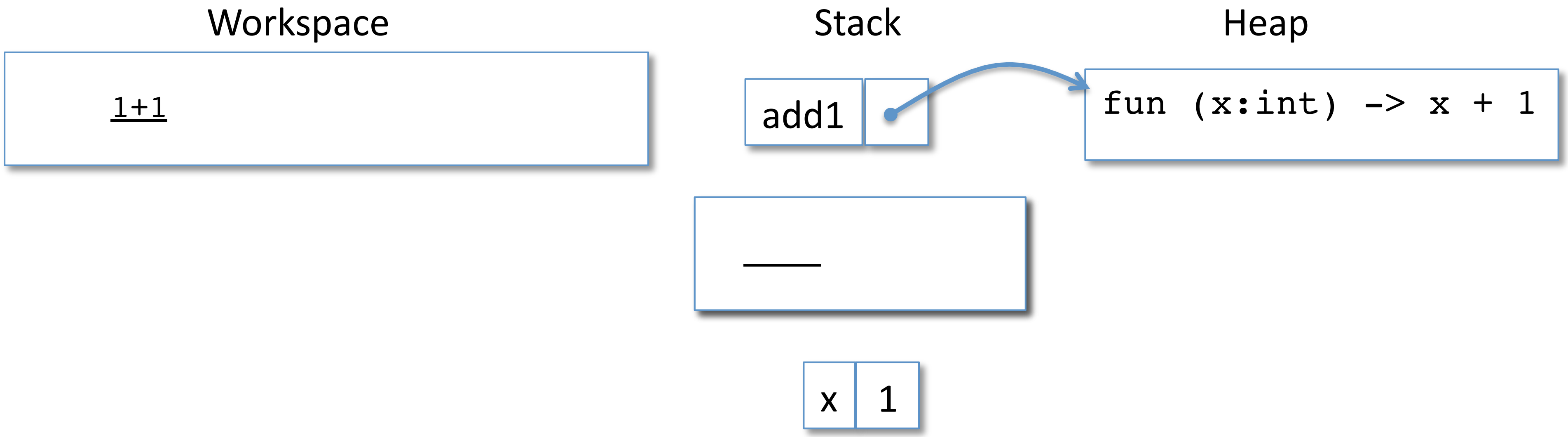
____

x 1

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

1+1

Stack

add1

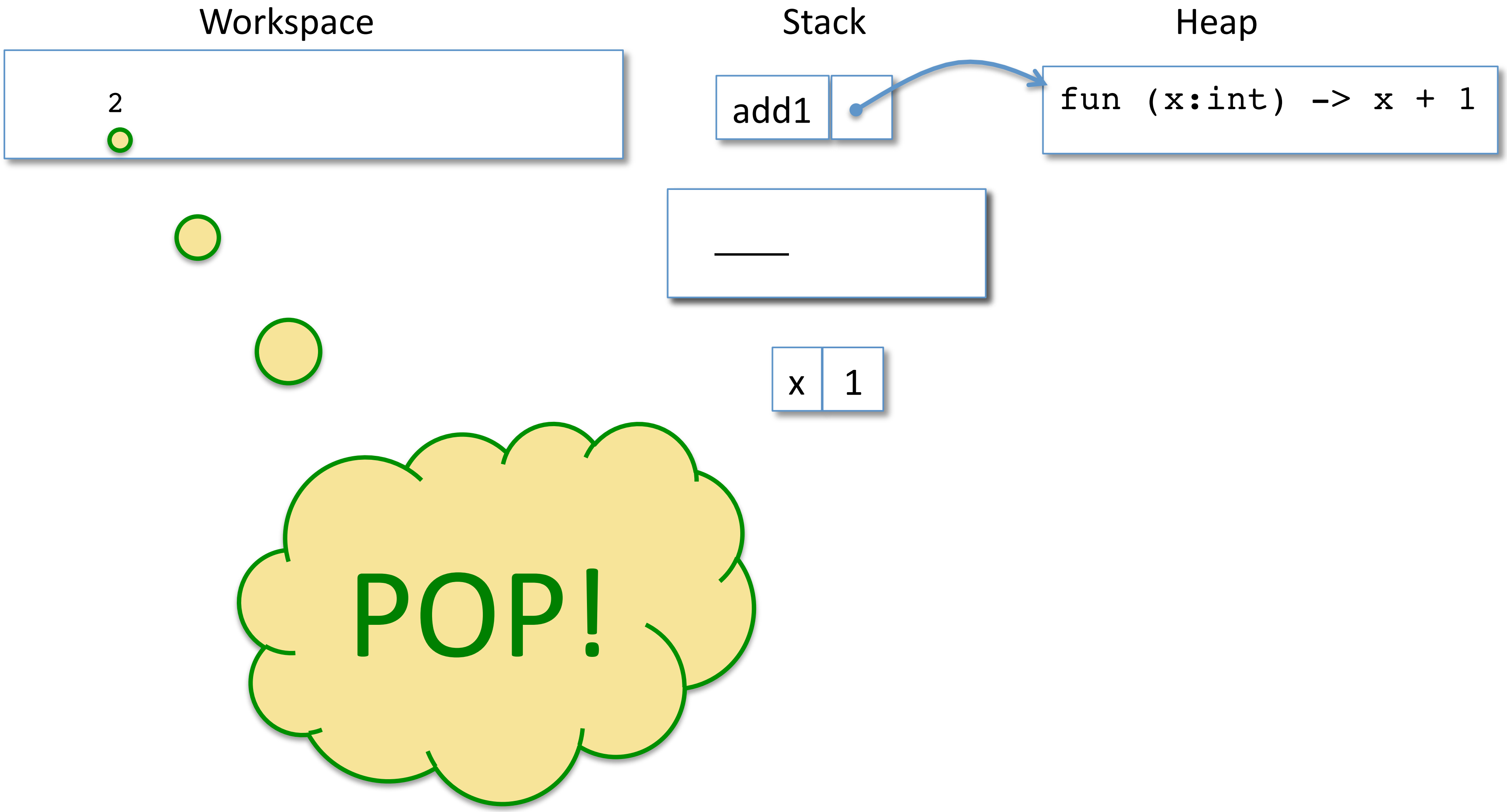
____

x 1

Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Workspace

2

Stack

add1

x

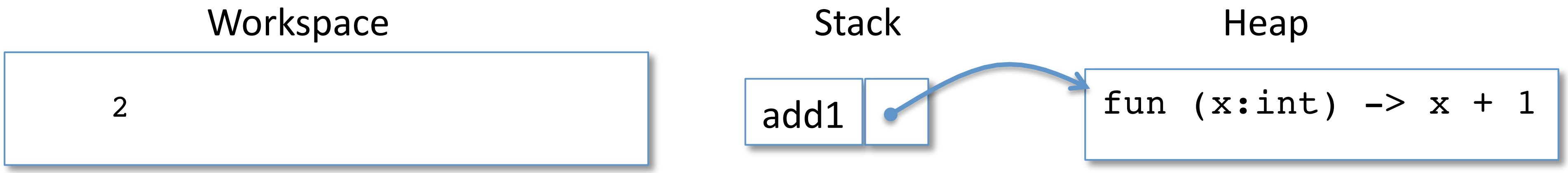
1

Heap

```
fun (x:int) -> x + 1
```

POP!

Function Simplification



Workspace

```
2
```

Stack

add1

Heap

```
fun (x:int) -> x + 1
```

DONE!

Simplifying Functions

- A function definition “let rec f (x_1:t_1)…(x_n:t_n) = e in body” is always ready.
  - It is simplified by replacing it with “let f = fun (x:t_1)…(x:t_n) = e in body”

- A function “fun (x_1:t_1)…(x_n:t_n) = e” is always ready.
  - It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.

- A function *call* is ready if the function and its arguments are all values
  - it is simplified by
    - saving the current workspace contents on the stack
    - adding bindings for the function’s parameter variables (to the actual argument values) to the end of the stack
    - copying the function’s body to the workspace

Function Completion

When the workspace contains just a single value, we *pop the stack* by removing everything back to (and including) the last saved workspace contents.

The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

If there aren't any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.